Guide de bonnes pratiques du développement logiciel

Christophe Couronne

GREYC, 13 octobre 2014

Résumé

Ce document est un recueil des bonnes et mauvaises pratiques de développement.

La plupart des pratiques présentées dans ce document ont fait l'objet d'un recensement sélectif parmi des référentiels connus dont la liste figure en bibliographie. Les autres relèvent de mon expérience et sont, par nature, empiriques.

Il s'agit d'un document de travail participatif dont le contenu est destiné à évoluer suivant une démarche continue d'amélioration des méthodes de développement.

Les pratiques présentées dans ce document sont classées en fonction de leurs portées (style de code, codage, conception...) ou par catégories arbitraires (Java, Java Enterprise Edition (JEE), PHP...).

Table des matières

1 Pratiques Générales

Cette partie du document s'intéresse aux règles transverses qui s'appliquent, peu ou proue, à tous les langages objets.

On y trouve des règles relatives :
— au style de code ;
— au codage ;
— à la conception ;
— au couplage.

Ces règles s'appliquent aussi bien à Java, qu'à PHP ou C++.

1.1 Style de code

Les règles présentées dans ce paragraphe touchent principalement au style du code. Elles formalisent quelques pratiques d'écriture avec pour objectif d'en faciliter la lecture.

1.1.1 Indentation du code

Il est préférable, pour la quasi totalité des langages objets, d'utiliser une indentation de 4 espaces, sans tabulation [doc14b] [Cor14].

Ceci permet d'éviter les problèmes avec les fichiers *diff*, les patches, l'historique CVS et les annotations.

1.1.2 Accolades autour des structures de contrôles

Il est préférable de toujours mettre les accolades autour des structures de contrôle `if`, `while`, `for`, `else`...

Cela facilite la lecture et lève toute ambiguïté.

```
/*
 * Mauvaise pratique
 */
if (isEnabled())
    System.out.println("Enabled !");
```

```
/*
 * Bonne pratique
 */
if (isEnabled()) {
    System.out.println("Enabled ! ");
}
```

1.1.3 Éviter les opérateurs ternaires

Les opérateurs ternaires sont jugés difficiles à lire par certains programmeurs. On préfère les éviter, d'autant plus qu'ils sont imbricables les uns dans les autres.

```
/*
 * Mauvaise pratique
 */
final Integer result = (isEnabled())
                           ? getValidCode()
                           : (isBusy())
                               ? getBusyCode()
                               : getErrorCode();
```

```
/*
 * Bonne pratique
 */

Integer result =  getErrorCode();

if (isEnabled()) {
    result = getValidCode();
}

if (isBusy()) {
    result = getBusyCode();
}
```

1.2 Métriques

Quelques mesures destinées à fournir des indicateurs de la qualité logicielle.

1.2.1 Longueur de classe

Les classes trop longues en font trop [Gri14].

Il s'agit de les restructurer afin qu'elles soient d'une longueur raisonnable.
On procède alors à une opération de réusinage.

La limite communément admise est de 1000 lignes par classe grand maximum.

```
/*
 * Mauvaise pratique (java)
 */

public class Foo {
    // 2000 lines of code
}
```

```
/*
 * Bonne pratique (java)
 */

public class Foo {
    // 1000 lines of code
}

public class Bar {
    // 1000 lines of code
}
```

1.2.2 Longueur des méthodes

Quand les méthodes sont excessivement longues, l'excès d'information provoque une perte d'attention du lecteur [Gri14].

Restructurer votre code en créant, par exemple, d'autres méthodes ou d'autres classes d'une longueur raisonnable.

Limite communément admise : maximum 100 lignes par méthode.

```
/*
 * Mauvaise pratique (java)
 */

/*
 * http://pmd.sourceforge.net/[...]#ExcessiveMethodLength
 */

public void doSomething() {

    System.out.println("Hello world!");

    // 200 copies omitted for brevity.
}
```

```
/*
 * Bonne pratique (java)
 */

public void doSomething() {

    System.out.println("Hello world!");

    // 100 copies omitted for brevity.

    doSubRoutine();
}

public void doSubRoutine() {

    // 100 copies omitted for brevity.

}
```

1.2.3 Nombre de paramètres

Une longue liste de paramètres indique que de nouveaux objets devraient être créés pour les encapsuler [Gri14].

Essayez de grouper les paramètres entre eux.

```
/*
 * Mauvaise pratique (java)
 *
 * http://pmd.sourceforge.net/[...]#ExcessiveParameterList
 */

public class Foo {
  public void addData(
    int p0, int p1, int p2, int p3, int p4, int p5,
    int p5, int p6, int p7, int p8, int p9, int p10) {
    /*
     * method body
     */
  }
}
```

```
/*
 * Bonne pratique (java)
 */

public class DataDto {

    private int p0;

    private int p1;

    private int p2;

    private int p3;

    private int p4;

    private int p5;

    private int p6;

    private int p7;

    private int p8;

    private int p9;

    private int p10;
```

```
    /*
     * 10 getters and setters omitted.
     */
}

...

public class Foo {

  public void addData(final DataDto dto) {
    /*
     * method body
     */
  }

}
```

1.2.4 Nombre de champs

Les classes qui ont trop de champs devraient être réusinées pour en avoir moins [Gri14].

Il est possible d'utiliser des objets imbriqués qui groupent une partie de l'information. Par exemple, une classe avec des champs ville / état / code postal pourrait, à la place, avoir un seul champ "Adresse".

La limite communément admise est de 15 champs par classe.

```
/*
 * Mauvaise pratique (java)
 *
 * http://pmd.sourceforge.net/[...]/codesize.html#TooManyFields
 */
public class Person {

   private String one;
   private int two;
   private int three;

   /*
    * [... many more public fields ...]
    */

   private String city;
   private String state;
   private String zip;

   /*
    * code omitted
    */
}
```

```
/*
 * Bonne pratique (java)
 */

public class Adresse {

    private String city;
    private String state;
    private String zip;

    /*
     * 3 getters and 3 setters omitted.
     */
}

/*
 * http://pmd.sourceforge.net/[...]/codesize.html#TooManyFields
```

```
 */
public class Person {

   private String one;
   private int two;
   private int three;

   /*
    * [... many more public fields ...]
    */

   private Adresse adresse;

   /*
    * code omitted
    */
}
```

1.2.5 Nombre de méthodes

Une classe avec de trop nombreuses méthodes est une bonne cible de réusinage [Gri14].

Cela permet de réduire sa complexité et d'avoir des objets d'une granularité plus fine.

Limite communément admise : 10 méthodes par classe.

```
/*
 * Mauvaise pratique (java)
 */
public class Foo
{
    public void foo1() {
        // code omitted
    }

    public void foo2() {
        // code omitted
    }

    public void foo3() {
        // code omitted
    }

    /*
     * 16 methods omitted
     */

    public void foo20() {
        // code omitted
    }
}
```

```
/*
 * Bonne pratique (java)
 */
public class Foo
{
    public void foo1() {
        // code omitted
    }

    public void foo2() {
        // code omitted
    }

    public void foo3() {
        // code omitted
    }
```

```
    /*
     * 6 methods omitted
     */

    public void foo10() {
        // code omitted
    }
}

public class Bar
{
    public void bar1() {
        // code omitted
    }

    public void bar2() {
        // code omitted
    }

    public void bar3() {
        // code omitted
    }

    /*
     * 6 methods omitted
     */

    public void bar10() {
        // code omitted
    }
}
```

1.2.6 Complexité cyclomatique

Il s'agit d'un concept simple qui est bien documenté dans PHPMD [Mod13].

On compte certaines instructions. On commence par 1 point à la déclaration de la fonction. Il faut ensuite ajouter 1 pour chaque `if`, `while`, `for` et `case`. Par exemple, la fonction ci-après possède une complexité cyclomatique de 12.

Le seuil standard de cette métrique est de 10 points. Si vous avez une fonction avec une complexité supérieure à 10, vous devez essayer de la réusiner.

```
1  public function example()  {
2    if ($a == $b)  {
3        if ($a1 == $b1) {
             fiddle();
4        } else if ($a2 == $b2) {
             fiddle();
         }  else {
             fiddle();
         }
5    } else if ($c == $d) {
6        while ($c == $d) {
             fiddle();
         }
7     } else if ($e == $f) {
8        for ($n = 0; $n > $h; $n++) {
             fiddle();
         }
     } else{
         switch ($z) {
9            case 1:
                 fiddle();
                 break;
10           case 2:
                 fiddle();
                 break;
11           case 3:
                 fiddle();
                 break;
12           default:
                 fiddle();
                 break;
         }
     }
}
```

1.2.7 Complexité NPATH

La complexité NPath est le nombre de chemins qui constituent le flux d'une fonction [Mod13]. Prenons en exemple la fonction suivante.

```
function foo($a, $b) {
    if ($a > 10) {
        echo 1;
    } else {
        echo 2;
    }
    if ($a > $b) {
        echo 3;
    } else {
        echo 4;
    }
}
```

Il y a 2 instructions `if` avec 2 sorties possibles chacune. On a donc 4 sorties possibles (2 * 2 = 4). Cela signifie que la complexité npath est de 4. Si nous avions une autre instruction `if`, nous aurions une complexité de 8, car (2 * 2 * 2 = 8).

La complexité NPath est exponentielle et peut facilement devenir incontrôlable, dans du vieux code. Ne soyez pas surpris si vous trouvez des fonctions avec une complexité supérieure à 100 000. La valeur par défaut du seuil de cette métrique est de 200. Vous devez rester sous cette valeur.

1.3 Codage

Les règles présentées dans ce paragraphe ciblent l'écriture même du code dans sa logique intrinsèque.

1.3.1 Limiter autant que possible les instructions `return` par fonction

Dans l'idéal, une méthode/fonction ne devrait avoir qu'un et un seul point de sortie. Ce devrait être sa dernière instruction.

L'instruction `return` fait partie des *jump statements*. Il y a 3 *jump statements* en java : `continue`, `break` et `return`. Ces instructions ont une action similaire à celle des instructions `goto` des langages BASIC ou Fortran : elles transfèrent le contrôle à un autre endroit du programme.

Pour limiter la complexité, on cherche à minimiser l'usage de ces instructions.

Cette règle fait partie des pratiques de développement dites de « programmation structurée » [ll14].

```
/*
 * Mauvaise pratique (Java).
 */
public class OneReturnOnly1 {
  public String foo(int x) {
    if (x > 0) {
      return "hey";   // oops, multiple exit points!
    }
    return "hi";
  }
}
```

```
/*
 * Bonne pratique (Java).
 */

public class OneReturnOnly1 {
  String result = "hi";
  public String foo(int x) {
    if (x > 0) {
      result = "hey";
    }
    return result;
  }
}
```

1.3.2 Éviter les instructions "en cascade"

Les instructions en cascade sont sources d'erreurs et compliquent inutilement le débogage des programmes.

Les piles d'appels et les erreurs de compilations sont bien souvent imprécises lorsqu'on ne respecte pas cette règle.

On ne peut simplement pas déterminer quelle instruction provoque l'exception.

```
/*
 * Mauvaise pratique (PHP).
 */

$message = Swift_Message::newInstance()
->setCharset(self::$configs['charset'])
->setSubject($this->subject)
->setFrom($assocAdressesFrom)
->setTo($assocAdressesTo)
->setBody($messageHtml, 'text/html')
->addPart(strip_tags($messageHtml), 'text/plain')
;
```

```
/*
 * Bonne pratique (PHP).
 *
 * Dans cet exemple, chaque instruction est décomposée.
 *
 * En cas d'erreurs, la pile d'appel est précise.
 */
$message = Swift_Message::newInstance();
$charset = self::$configs['charset'];
$part = strip_tags($messageHtml);
$message->setCharset($charset);
$message->setSubject($this->subject);
$message->setFrom($assocAdressesFrom);
$message->setTo($assocAdressesTo);
$message->setBody($messageHtml, 'text/html');
$message->addPart($part, 'text/plain');
```

```
/*
 * Mauvaise pratique (Java).
 */
builder.setLength(0);
builder.append("Switching file [")
```

```
        .append(nomFichier)
        .append("] in status WAITING_FOR_PROCESSING");

...

List<EtablissementUna> results = (List<EtablissementUna>) sessionFactory
      .getCurrentSession()
      .createCriteria(
          "fr.univbordeaux.[...].EtablissementUna")
      .add(create(instance)).list();
```

```
/*
 * Bonne pratique (Java).
 */
builder.setLength(0);
builder.append("Switching file [");
builder.append(nomFichier);
builder.append("] in status WAITING_FOR_PROCESSING");

...

final Session = sessionFactory.getCurrentSession();

final Criteria criteria = session.createCriteria(
    "fr.univbordeaux.[...].EtablissementUna");

final Criterion criterion = create(instance);

criteria.add(criterion);

final List<EtablissementUna> results = criteria.list();
```

1.3.3 Éviter les instructions `break`

L'instruction `break` fait partie des « jump statements ». Il y a 3 « jump statements » en java : `continue`, `break` et `return`. Ces instructions ont une action similaire à celle des instructions `goto` des langages BASIC ou Fortran : elles transfèrent le contrôle à un autre endroit du programme.

Pour limiter la complexité, on cherche à minimiser l'usage de ces instructions.

Cette règle fait partie des pratiques de développement dites de « programmation structurée » [ll14].

```
/*
 * Mauvaise pratique (Java).
 */

for(int i = 0;i<4;i++)
{
    if(i == 2) {
        break;
    }
    System.out.println("i = " + i);
}
```

```
/*
 * Bonne pratique (Java).
 */

for(int i = 0;i < 2;i++)
{
    System.out.println("i = " + i);
}
```

1.3.4 Éviter les instructions `continue`

L'instruction `continue` fait partie des « jump statements ». Il y a 3 « jump statements » en java : `continue`, `break` et `return`. Ces instructions ont une action similaire à celle des instructions `goto` des langages BASIC ou Fortran : elles transfèrent le contrôle à un autre endroit du programme.

Pour limiter la complexité, on cherche à minimiser l'usage de ces instructions.

Cette règle fait partie des pratiques de développement dites de « programmation structurée » [ll14].

```
for(int i = 0;i<4;i++)
{
    if(i == 2) {
        continue;
    }
    System.out.println("i = " + i);
}
```

```
for(int i = 0;i<4;i++)
{
    if(i != 2) {
        System.out.println("i = " + i);
    }
}
```

1.3.5 Éviter les conditionnelles négatives

Les expressions négatives sont plus difficiles à comprendre que les expressions positives.

Les expressions devraient être exprimées de manière positive [Mar09].

```
/*
 * Mauvaise pratique
 */

if (!buffer.shouldNotCompact())
```

```
/*
 * Bonne pratique
 */

if (buffer.shouldCompact())
```

1.3.6 Remplacer les nombres et les chaînes par des constantes

Il convient d'éviter de conserver les nombres et les chaînes de caractères en brut dans le code. La bonne pratique est d'utiliser des constantes nommées. Par exemple, le nombre 3 600 000 devrait être affecté à la constante `MILLIS_PER_HOUR` [Mar09].

```
/*
 * Mauvaise pratique (Java).
 */

final Calendar calendar = new GregorianCalendar();
final int dayOfMonth = calendar.get(5);
...
final new Bear("Teddy");
```

```
/*
 * Bonne pratique (Java).
 */

final Calendar calendar = new GregorianCalendar();
final int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);
...
final Bear bear = new Bear(Bear.NAME_OF_TEDDY);
```

1.4 Règles de conception

1.4.1 Le patron de conception *singleton*

Si une classe ne contient que des méthodes statiques, il est préférable d'en faire un singleton [Gri14].

Il faut alors ajouter un constructeur `private` (i.e. *privé*) pour éviter toute instanciation exterieure à cette classe.

cf. `http://pmd.sourceforge.net/rules/design.html#UseSingleton`.

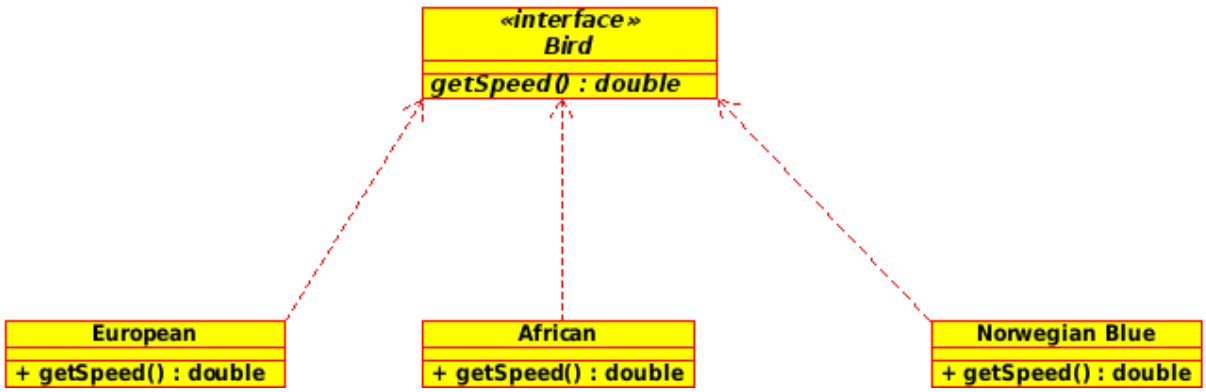
1.4.2 Substituer les instructions `switch` par du polymorphisme

On peut substituer les instructions `switch` lorsqu'elles sont utilisées pour choisir entre plusieurs comportements en fonction d'un type d'objet [Sou14].

Il s'agit de déplacer chaque composante de la condition dans une méthode surchargée au sein d'une sous classe. On rend la méthode originelle abstraite.

```
/*
 * Java, mauvaise pratique.
 *
 * http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism
 */
double getSpeed() {

   switch (_type) {

       case EUROPEAN:
          return getBaseSpeed();

       case AFRICAN:
          return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;

       case NORWEGIAN_BLUE:
          return (_isNailed) ? 0 : getBaseSpeed(_voltage);
   }

   throw new RuntimeException ("Should be unreachable");
}
```

Le polymorphisme permet d'utiliser une même interface avec différentes implémentations. Il s'agit de remplacer les instructions des différents cas fonctionnels, par des implémentations distinctes qui mettent en oeuvre la même interface de programmation.




```
public interface Bird
{
  double getSpeed();
}
```

```
public class European implements Bird
{

  public double getSpeed() {
    return getBaseSpeed();
  }

}
```

```
public class African implements Bird
{
    public double getSpeed() {
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
    }
}
```

```
public class NorwegianBlue implements Bird
{
    public double getSpeed() {

        int value = 0;

        if (!isNailed) {
            value = getBaseSpeed(voltage);
        }

        return value;
    }
}
```

```
public double getSpeed(type) {

   final Map<String, Bird> map = new HashMap<String, Bird>();

   map.put(EUROPEAN, new European());
   map.put(AFRICAN, new African());
   map.put(NORWEGIAN_BLUE, new NorwegianBlue());

   final Bird bird = map.get(type);

   return bird.getSpeed();
}
```

1.4.3 Éviter les champs protégés dans les classes `final`

Il est inutile d'utiliser des champs `protected` dans les classes `final` car elles ne peuvent pas être sous classées.

Clarifiez votre intention en utilisant le qualificatif `private` ou `friendly` à la place.

```
/*
 * Java, mauvaise pratique.
 */
public final class Bar {
  private int x;

  /*
   * Bar ne peut pas être sous classé, il s'agit donc d'un
   * champ private ou friendly.
   */
  protected int y;
  Bar() {}
}
```

```
/*
 * Java, bonne pratique.
 */
public final class Bar {
  private int x;

  /*
   * Bar ne peut pas être sous classé,
   * on rend le champ friendly.
   */
  int y;

  Bar() {}
}
```

```
/*
 * Java, bonne pratique.
 */
public final class Bar {
  private int x;

  /*
   * Bar ne peut pas être sous classé,
   * on rend le champ private.
   */
  private int y;
  Bar() {}
```

```
}
```

1.5 Couplage

Ces règles servent à réduire le couplage entre les différents composants des applications.

1.5.1 Typer par des interfaces

Les références de types d'implémentation complexifient la mise en oeuvre d'évolutions futures [Gri14]. Typer par des interfaces fournit beaucoup plus de flexibilité. Il est alors possible de changer facilement l'implémentation.

```
/*
 * Java, mauvaise pratique.
 *
 * http://pmd.sourceforge.net/[...]#LooseCoupling
 */

// sub-optimal approach
private ArrayList list = new ArrayList();

public HashSet getFoo() {
        return new HashSet();
}
```

```
/*
 * Java, bonne pratique.
 *
 * http://pmd.sourceforge.net/[...]#LooseCoupling
 */

// preferred approach
private List list = new ArrayList();

public Set getFoo() {
        return new HashSet();
}
```

2 Java

2.1 Documentation du code

2.1.1 Classes

La déclaration d'une classe doit être précédée d'un commentaire javadoc ou doxygen destiné à expliquer son fonctionnement.

```
package fr.unicaen.reactor;

/**
 * Une classe qui met en oeuvre le patron de conception « Reactor » afin de
 * gérer les traitements associés aux évènements métier.
 *
 * @author Christophe
 * @see Handler
 *
 * @param E le type d'évènement que l'on traite.
 * @param T le type qui identifie le callback dans le reacteur.
 */
public class Reactor<T,E>
{
    // skipped code
}
```

2.1.2 Methodes

La déclaration d'une méthode, même privée, doit faire l'objet d'un commentaire explicite qui expose :
— Son fonctionnement, c'est à dire ce qu'elle fait ;
— Le type et le sens de chaque paramètre attendu ;
— Si la méthode est une fonction (et non une procédure), le commentaire doit exprimer le type d'objet retourné et sa signification.

```
package fr.unicaen.reactor;

/**
 * Une interface pour les « callbacks » du reacteur.
 *
 * @author Christophe
 *
 * @see Reactor
 *
 * @param E le type d'évènement que l'on traite.
 * @param T le type qui identifiee le callback.
 */
public interface Handler<T,E>
{
    /**
     * Cette méthode donne le type "identifiant" qui
     * caractérise cet objet au sein du réacteur.
     *
     * @return le type sous forme de chaîne de caractère
     */
    T getHandlerType();

    /**
     * Traitement associé au type déclaré ci-dessus.
     *
     * {@link #getHandlerType() getHandlerType()}
     */
     void handleEvent(E event);
}
```

2.1.3 Héritage

Pour la documentation des méthodes d'implémentation, on peut, lorsqu'il s'agit d'une mise en oeuvre d'une méthode d'interface ou d'une méthode abstraite, éviter la redondance des commentaires à l'aide d'un commentaire non-javadoc clairement étiqueté.

```
package fr.unicaen.reactor;

/**
 * Une implementation de l'interface Handler pour le reactor.
 *
 * @author Christophe
 *
 * @see Reactor
 * @see Handler
 *
 * @see Caddie
 *
 * @param E le type d'évènement que l'on traite.
 * @param T le type qui identifie le callback dans le reacteur.
 */
public class HandleTeddyBear implements Handler<String, Caddie>
{
    /**
     * Une constante pour la clé (T) du reacteur.
     */
     private static final String TEDDY_BEAR_TYPE = "TeddyBear";

     /*
      *(non-Javadoc)
      *
      * @see fr.unicaen.reactor.Handler#getHandlerType()
      */
     public String getHandlerType() {
         return TEDDY_BEAR_TYPE;
     }

     /*
      *(non-Javadoc)
      *
      * @see fr.unicaen.reactor.Handler#getHandleEvent()
      */
     public void handleEvent(Caddie caddie) {
         caddie.add(new TeddyBear());
     }
}
```

2.1.4 Attributs

La déclaration d'un attribut ou d'une constante doit systématiquement faire l'objet d'un commentaire javadoc ou doxygen destiné à en expliciter la présence.

```
/**
 * Le logger log4j.
 */
private static final LOGGER = Logger.getLogger(Reactor.class);

/**
 * Map destinée à recevoir les handleurs.
 */
private Map<String, Caddie> handlers = new HashedMap<String, Caddie>();
```

2.2 Règles de conception et de codage

2.2.1 Éviter les retours `null`

Quand on retourne `null`, on se crée du travail en imposant à l'appelant de faire un contrôle des données. Un contrôle qui manque sur le `null` et l'application est plantée [Mar09].

Si vous souhaitez retourner `null`, vous devriez lever une exception ou retourner un objet vide à la place.

Si vous appelez une méthode d'une API tierce qui retourne `null`, il faut envelopper cette méthode dans une autre méthode qui lève une exception ou retourne un objet vide.

Dans de nombreux cas, les objets vide sont des solutions pratiques.

Si l'on étudie le code suivant :

```
/*
 * Clean Code, A Handbook of Agile Software Craftsmanship
 */

final List<Employee> employees = getEmployees();
if (employees != null) {
  for (final Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

`getEmployees()` peut retourner `null`, mais le doit-il vraiment ? si on change `getEmployee()` afin qu'il retourne une liste vide, on peut nettoyer le code ainsi :

```
/*
 * Clean Code, A Handbook of Agile Software Craftsmanship
 */

final List<Employee> employees = getEmployees();
for (final Employee e : employees) {
  totalPay += e.getPay();
}
```

L'API java propose une méthode `Collections.emptyList()` qui retourne une liste immuable prédéfinie que l'on peut utiliser dans ce cas précis :

```
/*
 * Clean Code, A Handbook of Agile Software Craftsmanship
 */

public List<Employee> getEmployees() {
  if ( .. there are no employees .. ) {
    return Collections.emptyList();
  }
}
```

Si vous codez de cette manière, vous minimisez le risque `NullPointerException` et le code est bien meilleur.

2.2.2 Éviter les passages de paramètres `null`

Une méthode qui retourne `null` est mauvaise, mais passer `null` en paramètre de méthode est encore pire. Sauf si vous travaillez sur une API qui attend que vous passiez `null`, il faut éviter autant que possible de passer `null` dans le code [Mar09].

2.2.3 La variable devrait être déclarée `final`

Une variable assignée une seule fois doit être déclarée `final` [Gri14].

De même, on ne réassigne jamais les paramètres d'une fonction. Ceux-ci devraient donc toujours être notés `final`.

```
/*
 * Mauvaise pratique Java
 *
 * PMD example.
 *

 http://pmd.sourceforge.net/[...]#MethodArgumentCouldBeFinal
 http://pmd.sourceforge.net/[...]#LocalVariableCouldBeFinal

 */

public class Bar {

    public void foo1 (String param) {

        /*
         * do stuff with param never assigning it
         */

        // skipped code
    }

    public void foo () {
        /*
         * if txtA and txtB will not be assigned again.
         */
        String txtA = "a";
        String txtB = "b";

        // skipped code
    }
}
```

```
/*
 * Bonne pratique Java
 *
 * PMD example.
 *

 http://pmd.sourceforge.net/[...]#MethodArgumentCouldBeFinal
 http://pmd.sourceforge.net/[...]#LocalVariableCouldBeFinal
```

```
 */

public class Bar {

    public void foo2 (final String param) {
        /*
         * better, do stuff with param never assigning it
         */

        // skipped code
    }

    public void foo () {

        /*
         * It is better to do this.
         */

        final String txtA = "a";
        final String txtB = "b";

        // skipped code
    }
}
```

2.2.4 Utiliser ArrayList au lieu de Vector

La classe `Vector` de l'API Java standard prend en charge les logiques d'exclusion mutuelle propres aux contextes concurrentiels. On dit qu'elle est *thread safe*.

`ArrayList` est une meilleure implémentation de l'interface *Collection* si l'on ne travaille pas dans un contexte concurrentiel.

On gagne en performances.

```
/*
 * Mauvaise pratique (Java)
 *
 * On utilise Vector dans un context non concurrentiel.
 */

public class SimpleTest extends TestCase {

    public void testX() {

        final Collection c1 = new Vector();

        // skipped code
    }

}
```

```
/*
 * Bonne pratique (Java)
 *
 * On utilise ArrayList dans un context non concurrentiel.
 */

public class SimpleTest extends TestCase {

    public void testX() {

        final Collection c1 = new ArrayList();

        // skipped code
    }
}
```

2.2.5 Utiliser StringBuilder au lieu de StringBuffer

L'utilisation de l'opérateur `+=` pour les chaînes de caractères oblige la machine virtuelle à créer un `StringBuilder` interne à chaque opération [Gri14]. Il est préférable d'utiliser un `StringBuilder` unique et explicite.

La classe `StringBuffer` de l'API Java standard prend en charge les logiques d'exclusion mutuelle propres aux accès concurrents des contextes concurrentiels. On dit qu'elle est *thread-safe*.

On préfère l'usage de `StringBuilder` à celle du `StringBuffer` si l'on ne travaille pas dans un contexte concurrentiel.

On gagne alors en performances.

```
/*
 * Mauvaise pratique (Java)
 *
 * On utilise += sur une chaine de caractère.
 */
public class Foo {

  public void bar() {

    String a;

    a = "foo";
    a += " bar";   // la VM crée un StringBuilder.
    a += " bar++"; // la VM crée un autre StringBuilder.

    // skipped code

  }

}
```

```
/*
 * Mauvaise pratique (Java)
 *
 * On utilise StringBuffer dans un contexte non concurrentiel.
 */
public class Foo {

  public void bar() {

    final StringBuffer buffer = new StringBuffer();

    buffer.append("foo");
    buffer.append("bar");

    // skipped code
```

```
  }

}
```

```
/*
 * Bonne pratique (Java)
 *
 * On utilise StringBuilder dans un contexte non concurrentiel.
 */
public class Foo {

  public void bar() {

    final StringBuilder builder = new StringBuilder();

    builder.append("foo");
    builder.append("bar");

    // skipped code
  }

}
```

2.2.6 Utiliser des énumérations en lieu et place des listes de constantes

Les énumérations sont une évolution de java 5. Elles proposent une alternative aux listes de constantes dans la mesure où celles-ci sont homogènes (ie. de même type). L'énumération devient alors un type d'objet à part entière au même titre qu'une classe.

```
/*
 * Mauvaise pratique java.
 */
public static final class CaddyItemTypes {
    public static final String BEAR = "Bear";
    public static final String RABBIT = "Rabbit";
}
```

```
/*
 * Bonne pratique java.
 */
public enum CaddyItemType {

    BEAR("Bear"),

    RABBIT("Rabbit");

    private String value;

    private CaddyItemType(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

2.2.7 Éviter de proposer un attribut d'instance de visibilité publique

D'une manière générale, une classe est responsable de ses attributs et elle ne doit jamais proposer d'accès "public" sur ceux-ci.

L'usage du qualifier "protected" est controversé car les classes qui héritent peuvent avoir besoin d'accéder au champ de la super classe.

Toutefois, il ne faut pas oublier qu'un qualifier "protected" autorise les classes du paquetage à accéder au champ, ce qui est finalement très ouvert.

La pratique courante est de protéger les champs à l'aide du qualifier "private" et de proposer, au cas par cas, des méthodes dites getters et setters pour y accéder si besoin est.

La classe est alors garante de l'intégrité de ses données.

2.2.8 Éviter de qualifier la visibilité dans les interfaces

Une interface spécifie un ensemble de méthodes abstraites.

Il s'agit d'un type très abstrait qui ne contient que le strict nécessaire à la mise en oeuvre ultérieure par des classes dérivées.

Java autorise, pour chaque méthode abstraite de déclarer (ou non) sa visibilité. Il s'agit conceptuellement d'un détail d'implémentation qui est souvent considéré comme inutile pour la lecture des interfaces de programmation.

On préfère écrire les interfaces sans les informations relatives à la visibilité.

Quel intérêt de proposer une interface de service "protected" ou "private" ?

```
/*
 * Java, mauvaise pratique
 */
public interface Bear
{
    public void eat(Salmon salmon);

    public void walkAlong(River river);
}
```

```
/*
 * Java, bonne pratique
 */
public interface Bear
{
    // le qualifier public disparait, il
    // sera déclaré dans la classe dérivé.

    void eat(Salmon salmon);

    void walkAlong(River river);
}
```

2.2.9 Qualifier "public", "protected" et "private" les méthodes des classes

Une règle de bon sens. On évite d'utiliser l'absence de qualifier, c'est à dire le qualifier "friendly" qui s'approche de "protected" mais sans l'accès aux sous méthodes héritées.

Il est préférable d'expliciter clairement la visibilité des méthodes de chaque classe parmi les 3 qualifiers suivants :

— public ;
— private ;
— protected.

3 Java Enterprise

3.1 Traitement par lots

3.1.1 Pré-allocation des séquences

Les générateurs d'identifiants de JPA et d'hibernate utilisent par défaut une pré-allocation de séquence de taille 1[Sut11].

Dans cette configuration, chaque instruction `insert` ou `update` provoque l'appel à une autre instruction `select` qui sert à récupérer la nouvelle valeur de la séquence.

En conséquence, les accès sont doublés en base de données lors des traitements par lots.

Cette configuration est excessive. Si l'on modifie ce critère pour 500 ou 1000, on réduit de moitié les accès dans les procédures de traitement par lots.

Les identifiants ne sont cependant plus contigus en base, mais les performances sont au rendez-vous.

```
<hibernate-mapping>
    <class name="fr.univbordeaux.[...].IndividuUna"
           table="individu_una">

        <id name="idIndividuUna" type="int">
            <column name="id_individu_una" />
            <generator class="sequence">
                <param name="sequence">seq_individu_una</param>

                <!--
                  - AllocationSize vaut maintenant 1000.
                  -->
                <param name="allocationSize">1000</param>

            </generator>
        </id>

        ...

    </class>
</hibernate-mapping>
```

3.1.2 Utiliser `hibernate.jdbc.batch_size`

Sans une configuration adéquate, JPA et hibernate ne sont pas prêts pour gérer correctement les traitements par lots.

Les nouvelles instances sont insérées dans le cache de second niveau[Doc14a] ce qui provoque généralement une `OutOfMemoryException` aux alentours du 50 000 ème objet traité.

On configure donc le framework afin qu'il place les objets dans le cache de premier niveau et ce, avec une fenêtre raisonnable (10-50)

```
hibernate.jdbc.batch_size 20
```

Lorsque vous rendez de nouveaux objets persistants ou lorsque vous les mettez à jour, vous devez régulièrement appeler `flush()` et puis `clear()` sur la session, pour contrôler la taille du cache de premier niveau[Doc14a].

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for (int i=0; i<100000; i++) {

    Customer customer = new Customer(.....);
    session.save(customer);

    if ( i % 20 == 0 ) {
        /*
         * 20, same as the JDBC batch size
         *
         * flush a batch of inserts and release memory:
         */
        session.flush();
        session.clear();
    }

}

tx.commit();
session.close();
```

3.2 Logging

3.2.1 Éviter le `println()`

Dans les applications d'entreprise, il convient d'éviter les instructions de type `System.out.println()`. Il est préférable d'utiliser un logger [Gri14].

```
/*
 * Java, mauvaise pratique
 */

class Foo {

    /**
     * Corps de la fonction.
     */
    public void testA () {
        System.out.println("Entering test");
    }
}
```

```
/*
 * Java, bonne pratique
 */

class Foo {

    /**
     * Le logger log4j.
     */
    public static final Logger LOGGER = Logger.getLogger(Foo.class.getName());

    /**
     * Corps de la fonction.
     */
    public void testA () {
        // Better use this
        LOGGER.fine("Entering test");
    }
}
```

3.2.2 Éviter le `printStackTrace()`

Dans les applications d'entreprise, évitez d'appeler printStackTrace(), il est préférable d'utiliser un logger.

On conserve ainsi les exceptions dans un log [Gri14].

```
public class Foo {
/*
 * Java enterprise, mauvaise pratique.
 */

/**
 * Corps de la fonction.
 */
public void bar() {

  try {
    // do something
  }
  catch (final Exception e) {
    e.printStackTrace();
  }
}
```

```
/*
 * Java, bonne pratique.
 */
public class Foo {

  /**
   * Le logger log4j.
   */
  private static final Logger LOGGER = Logger.getLogger(Foo.class.getName());

  /**
   * Corps de la fonction.
   */
  public void bar() {
    try {
      // do something
    }
    catch (final Exception e) {
      LOGGER.error("Uncaught exception", e);
    }
  }
}
```

3.2.3 Un logger par classe

Normallement, un seul et unique logger est utilisé pour une classe et il devrait être `private static final` [Gri14].

```
/*
 * Java, bonne pratique.
 */
public class Foo {

    /**
     * Le logger log4j.
     */
    private static final Logger LOGGER = Logger.getLogger(
        Foo.class.getName()
    );

    // skipped code
}
```

3.2.4 Bien logger un message en mode debug

On utilise `isDebugEnabled()` pour optimiser le traitement des messages en mode nominal. Quand le `debug` n'est pas activé, on gagne en temps de traitement : log4j ne cherche alors pas à calculer le niveau du message au regard de la configuration complète. Il se contente de dire si le *debug* est activé et ne cherche pas à étudier la pile des niveaux de logs autorisés.

On gagne en temps de traitement.

```
/*
 * Java, bonne pratique.
 */
public class Foo {

    /**
     * Le logger log4j.
     */
    private static final Logger LOGGER = Logger.getLogger(
        Foo.class.getName()
    );

    /**
     * function body.
     */
    public void bar(final String name) {

        if (LOGGER.isDebugEnabled()) {

            final StringBuilder builder = new StringBuilder();

            builder.append("name is [");
            builder.append(name);
            builder.append("]");

            final String message = builder.toString();

            LOGGER.debug(message);
        }
    }
}
```

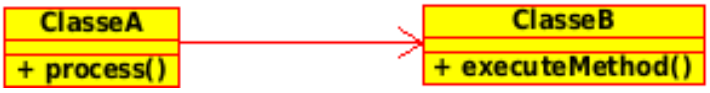
3.3 Couplage

3.3.1 Utiliser Spring et l'inversion de contrôle

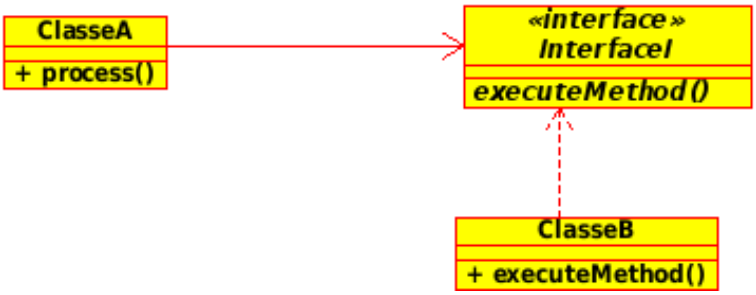
Spring est un conteneur léger d'application javaee. Il prend en charge l'instanciation des objets à l'aide d'un patron de conception qui réduit le couplage entre les classes : l'inversion de contrôle (IOC, Inversion Of Control).

L'inversion de contrôle remplace l'instanciation des dépendances par une injection de celles-ci à l'aide d'un *framework*, en l'occurence, spring.

Exemple : nous avons une dépendance directe d'une classe A vers une classe B.



Il s'agit d'introduire un typage par interfaces afin de réduire le couplage entre les classes A et B.



La classe A ne dépend donc plus de B directement, mais de I qui peut être implémenté par n'importe quelle autre classe.

Les dépendances sont instanciées et injectées par le *framework*.
Mise en oeuvre

```
public interface I
{

  void executeMethod();

}
```

```
public class A
{

   private I dependanceI;

   public A() {
   }

   public A(final I dependanceI) {
       this.dependanceI = dependanceI;
   }

   public void setDependanceI(final I dependanceI) {
       this.dependanceI = dependanceI;
   }

   public void process() {
       dependanceI.executeMethod();
   }

}
```

```
public class B implements I
{

    public B() {
    }

    public void executeMethod() {
        // skipped code
    }

}
```

Injection des dépendances à l'aide de Spring

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="beanB" class="fr.univbordeaux.ioc.B" />

    <bean name="beanA" class="fr.univbordeaux.ioc.A">
        <property name="dependancel" ref="beanB"/>
    </bean>

</beans>
```

4 PHP

Pratiques qui s'intéressent, quasi exclusivement, à la programmation en PHP.

Par nature, PHP est complaisant et permissif avec de nombreuses pratiques. Il convient de prendre quelques habitudes pour rendre le code moins complexe à analyser et éviter des problèmes de sécurité.

4.1 Développer en `E_ALL|E_STRICT`

E_ALL et E_STRICT sont d'une aide précieuse pour écrire un code de qualité.

— `E_ALL` affiche toutes les erreurs, les warnings ;
— `E_STRICT` affiche des suggestions pour améliorer l'interopérabilité et l'évolutivité du code.

Dans l'idéal, il faudrait toujours développer ainsi.

```
/*
 * Bonne pratique de développement : utiliser E_STRICT
 */

error_reporting(E_ALL | E_STRICT);
```

La fonction `error_reporting` est parfois désactivée, notamment si PHP s'exécute en *safe-mode*.

On peut, dans ce cas, forcer le mode `E_ALL | E_STRICT` dans un fichier `.htaccess` de la manière suivante :

```
php_value error_reporting 4095
```

On peut également ajouter les paramètres de configuration suivants pour forcer l'affichage d'un maximum d'informations.

```
php_flag display_errors On
php_flag track_errors On
php_flag html_errors On
php_flag report_memleaks On
php_flag display_startup_errors On
php_flag log_errors On
```

4.2 Utiliser les structures de contrôle du langage C

En PHP, il est possible d'écrire des structures de contrôle (`if`, `else`, `for`, `while`, `switch`) en respectant les styles des langages C ou BASIC.

Les structures de contrôle du langage C sont reprises par les langages C, Java, C# et C++. De ce fait, elles sont familières pour la plupart des développeurs. Les structures de contrôle en style BASIC sont à éviter.

```
/*
 * Mauvaise pratique : écrire les structures de contrôle dans
 * le style du langage BASIC.
 */
if ($a == 5):
    echo "a est égal à 5";
    echo "...";
else:
    echo "a n'est pas égal à 5";
endif;
```

```
/*
 * Bonne pratique : écrire les structures de contrôle dans le style
 * du langage C.
 */
if ($a == 5) {
    echo "a est égal à 5";
    echo "...";
}
else {
   echo "a n'est pas égal à 5";
}
```

4.3 Ne pas utiliser les *short open tags*

Les « *short open tags* » posent des problèmes de portabilité. Tous les serveurs ne les supportent pas. L'interpréteur PHP CLI les considère d'ailleurs comme des erreurs de syntaxe.

Il est préférable d'utiliser l'instruction complète `<?php`

```
<?

    /*
     * Mauvaise pratique (PHP)
     */

    echo 'PHP code written using short open tags';
?>
```

```
<?php

    /*
     * Bonne pratique (PHP)
     */

    echo 'PHP code written using full php tags';

?>
```

4.4 Constructeurs PHP5

Il est préférable de nommer les constructeurs des classes `__construct()`.

Les conventions de nommage de PHP4 sont toujours supportées pour des raisons de compatibilité ascendante.
Elles sont cependant déconseillées (dépréciées) dans la documentation de PHP.

```
/*
 * Mauvaise pratique (PHP)
 */

class DBConnect
{
    /**
     * Constructeur par défaut.
     *
     * Construit une instance de DBConnect.
     */
    public DBConnect() {
```

```
/*
 * Bonne pratique (PHP)
 */

class DBConnect
{
    /**
     * Constructeur par défaut.
     *
     * Construit une instance de DBConnect.
     */
    public __construct() {
```

4.5 Ne jamais utiliser de variables globales

Les variables globales peuvent être modifiées par l'interpréteur PHP depuis d'autres endroits du code, ce qui peut produire des effets de bord.

Il convient de garder des variables de portées locales.

Le code s'en trouve plus simple et plus propre. C'est une aide pour son débugage, sa relecture et sa pérénité.

D'une manière générale, lorsque le mot clé `global` est utilisé, on considère suspect le code et on procède à une opération de réusinage pour le corriger.

4.6 Éviter les conversions implicites

PHP peut convertir des types de manière implicite en silence et sans précautions.

C'est notamment le cas si l'on additionne une chaîne de caractères (qui représente un entier) avec un entier. La valeur de la chaîne de caractères est alors extraite puis convertie en entier.

Le développeur distrait n'a plus conscience du type de données qu'il manipule ! On prèfère évidemment une conversion explicite de type.

```
/*
 * Mauvaise pratique (PHP)
 */

/*
 * $groupe['count_etapes'] est une chaîne de caractère
 * qui représente un entier.
 *
 * On cherche ici à savoir si cette valeur est supérieure à zéro.
 *
 * Il faut la convertir en entier pour pouvoir tester sa
 * valeur numérique.
 */
$count_etapes = $groupe['count_etapes']
if (($count_etapes + 0) > 0) {
    ...
}
```

```
/*
 * Bonne pratique (PHP)
 */

/*
 * Pour convertir $count_etapes en entier, il est préférable
 * d'utiliser l'instruction intval().
 */
$count_etapes = intval($groupe['count_etapes']);
if ($count_etapes > 0) {
    ...
}
```

4.7 Toujours initialiser ses variables

Il n'est pas cohérent d'utiliser une variable sans l'avoir préalablement déclarée et affectée.

Le développeur paresseux peut être heureux car PHP va automatiquement déclarer ces variables pour lui. Charge au relecteur de trouver s'il s'agit ou non d'un oubli. Le code est brouillon et difficile à comprendre.

Il convient donc de toujours déclarer et initialiser ses variables de manière explicite.

```
/*
 * Mauvaise pratique (PHP)
 */

function retrieveValues() {
    /*
     * La méthode retrieveValuesImpl() renseigne le tableau
     * $data passé par référence.
     *
     * Dans cet exemple $data est déclaré de manière implicite.
     * C'est une mauvaise pratique, bien que PHP l'autorise.
     */
    retrieveValuesImpl($data);

    return $data;
}
```

```
/*
 * Bonne pratique (PHP)
 */

function retrieveValues() {
    /*
     * On déclare de manière explicite $data avant de
     * le passer à la fonction retrieveValuesImpl()
     *
     * C'est la bonne pratique.
     */
    $data = array();

    retrieveValuesImpl($data);

    return $data;
}
```

4.8 Concaténation d'une variable valeur NULL

Une variante de la règle présentée précédemment : PHP permet de concaténer une variable NULL avec une chaine de caractères. Le code ainsi créé est, là encore, confus et brouillon. Il est préférable d'initialiser la variable avant de la concaténer.

```
$where = NULL;
if (empty($cod_cge) === FALSE) {
    /*
     * Attention, on va ici concaténer une variable NULL
     * avec une chaîne de caractères !
     */
    $where .= "(ETP.cod_cge='";
    ...
}
```

```
/*
 * Bonne pratique de développement : initialiser
 * $where à une autre valeur que NULL :)
 */
$where = '';
if (empty($cod_cge) === FALSE) {
    /*
     * $where n'est pas NULL, on peut
     * concaténer les yeux fermés.
     */
    $where .= "(ETP.cod_cge='";
    ...
}
```

4.9 Éviter l'appel de fonctions dans le corps d'une déclaration de boucle `for`

Appeler une fonction dans le corps d'une déclaration de boucle `for` est inutilement couteux en calcul et peut être source d'erreurs. Il vaut mieux écrire une ligne de plus.

```
/*
 * Mauvaise pratique (PHP)
 */

for ($i = 0; $i < count($array); $i++) {
    ...
}
```

```
/*
 * Bonne pratique (PHP)
 */

$count = count($array);

for ($i=0; $i < $count; $i++) {
    ...
}
```

5 Python

5.1 Exécuter l'interpréteur avec l'option `-tt`

Il est préférable de lancer l'interpréteur avec l'option `-tt` et ce, afin de lever des erreurs si l'indentation n'est pas concistante.

Bibliographie

Références

[Cor14] Oracle Corp. Code conventions for the java programming language : Contents, 2014. http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

[Doc14a] Hibernate Community Documentation. Chapitre 14. traitement par lot, 2014. http://docs.jboss.org/hibernate/core/3.5/reference/fr-FR/html/batch.html.

[doc14b] The PEAR documentation. Indentation et longueur de lignes, 2014. http://pear.php.net/manual/fr/standards.indenting.php.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series, 1994.

[Gri14] Miguel Griffa. Pmd rulesets index, 2014. http://pmd.sourceforge.net/pmd-5.1.1/rules/index.html.

[Il14] Wikipedia l'encyclopédie libre. Programmation structurée – wikipédia, 2014. http://fr.wikipedia.org/wiki/Programmation_structur%C3%A9e.

[Mar09] Robert C. Martin. *Clean Code, A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.

[Mod13] Niklas Modess. Cyclomatic and npath complexity explained, 2013. http://codingswag.ghost.io/cyclomatic-and-npath-complexity-explained/.

[PHP14a] Manuel PHP. Php : Constructors and destructors - manual, 2014. http://php.net/manual/en/language.oop5.decon.php.

[PHP14b] Manuel PHP. Php : Syntaxe alternative - manual, 2014. http://php.net/manual/fr/control-structures.alternative-syntax.php.

[Sou14] SourceMaking. Replace conditional with polymorphism, 2014. http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism.

[Sut11] James Sutherland. Java persistence performance : How to improve jpa performance by 1 825 %, 2011. http://java-persistence-performance.blogspot.fr/2011/06/how-to-improve-jpa-performance-by-1825.html.

[Tea14] Checkstyle Development Team. checkstyle - available checks, 2014. http://checkstyle.sourceforge.net/availablechecks.html.